Informatique en CPGE (2018-2019) Algorithmique et programmation de base (partie 2)

S. B.

Lycée des EK

30 septembre 2018

Considérons l'instruction :

```
nombre = nombre +1
```

Cette instruction signifie : "évalue l'expression de droite et affecte la valeur obtenue à la variable de gauche". Ceci permet donc d'incrémenter une variable de une unité et s'utilise souvent dans des boucles.

On écrit aussi (un peu moins lisible mais plus rapide) :

```
nombre += 1
```

On utilise l'instruction "Tant que" (While). Cette instruction s'emploie pour répéter une suite d'instructions lorsque le nombre de répétitions est inconnu.

On utilise l'instruction "Tant que" (While). Cette instruction s'emploie pour répéter une suite d'instructions lorsque le nombre de répétitions est inconnu.
La suite d'instructions est répétée tant qu'une certaine condition est vraie : on commence donc par un test, si le test est `True` alors on parcourt le corps de la boucle une fois ; ensuite on réévalue le test.

On utilise l'instruction "Tant que" (While). Cette instruction s'emploie pour répéter une suite d'instructions lorsque le nombre de répétitions est inconnu.
La suite d'instructions est répétée tant qu'une certaine condition est vraie : on commence donc par un test, si le test est `True` alors on parcourt le corps de la boucle une fois ; ensuite on réévalue le test.
Ce processus est répété jusqu'à ce que le test soit `False` ; on passe alors à la suite du code.

Un exemple d'algorithme :

```
Variable x
x ...
Début Tant que <condition sur x>
        ... Instructions ...
Fin Tant que
```

L'implémentation en Python est :

```
x = ...
while (condition) :
        action1
        action2
        action3
```

La condition sur x peut être par exemple : (x < 100 and x > 10).

Attention à toujours vérifier que la boucle ne va pas se répéter sans fin.

Attention à toujours vérifier que la boucle ne va pas se répéter sans fin.
Si la valeur de x n'est pas modifiée dans le corps de la boucle, soit le test est toujours `False` et la boucle n'est jamais exécutée, soit toujours `True` et alors la boucle est exécutée sans fin.

Attention à toujours vérifier que la boucle ne va pas se répéter sans fin.
Si la valeur de x n'est pas modifiée dans le corps de la boucle, soit le test est toujours `False` et la boucle n'est jamais exécutée, soit toujours `True` et alors la boucle est exécutée sans fin.

Par exemple, l'algorithme suivant est-il correct ?

```
i = 1
Début Tant que i > 0
        i = i + 1
Fin Tant que
```

Que penser du programme suivant ? Et si on remplace la première ligne par `x=1` ?

```
x = 1.0
y = x + 1
while y - x == 1 :
        x = 10*x
        y = x + 1
print ('x =', x, 'y =', y)
```

Que penser du programme suivant ? Et si on remplace la première ligne par `x=1` ?

```
x = 1.0
y = x + 1
while y - x == 1 :
      x = 10*x
      y = x + 1
print ('x =', x, 'y =', y)
```

Si on exécute ce programme, on obtient `x=1e+16` et `y=1e+16`, soit 10^{16}. Le même programme exécuté sur une calculatrice donne 10^{12} ou 10^{14} ou . . . , suivant les modèles.

Explication : x est du type `float`. Or la taille et la précision des nombres de type `float` dans la machine est limitée ;

Explication : x est du type `float`. Or la taille et la précision des nombres de type `float` dans la machine est limitée ; donc il existe un grand nombre K, par exemple $K = 10^{16}$, tel que 1 est négligeable par rapport à K et alors $K + 1$ est égal à K (pour la machine !).

Explication : x est du type `float`. Or la taille et la précision des nombres de type `float` dans la machine est limitée ; donc il existe un grand nombre K, par exemple $K = 10^{16}$, tel que 1 est négligeable par rapport à K et alors $K + 1$ est égal à K (pour la machine !). La suite de nombres ainsi construite, 1, 2, 10, 11, 100, 101, ... est croissante et majorée par K ! Donc elle est convergente et le programme va bien se terminer.

Explication : x est du type `float`. Or la taille et la précision des nombres de type `float` dans la machine est limitée ; donc il existe un grand nombre K, par exemple $K = 10^{16}$, tel que 1 est négligeable par rapport à K et alors $K + 1$ est égal à K (pour la machine !). La suite de nombres ainsi construite, 1, 2, 10, 11, 100, 101, ... est croissante et majorée par K ! Donc elle est convergente et le programme va bien se terminer.

Par contre si x et y sont du type `int`, le programme ne se termine pas car en Python les entiers sont illimités.

Application à la dichotomie

Application à la dichotomie

On cherche un nombre qui appartient à un ensemble de nombres ordonnés.

Application à la dichotomie

On cherche un nombre qui appartient à un ensemble de nombres ordonnés. La dichotomie consiste à partager l'ensemble en deux ensembles de même taille puis à tester auquel des deux ensembles le nombre appartient ;

Application à la dichotomie

On cherche un nombre qui appartient à un ensemble de nombres ordonnés. La dichotomie consiste à partager l'ensemble en deux ensembles de même taille puis à tester auquel des deux ensembles le nombre appartient ; on recommence alors avec le nouvel ensemble et on reproduit ce processus **tant que** la taille de l'ensemble est supérieure à la précision souhaitée.

Application à la dichotomie

On cherche un nombre qui appartient à un ensemble de nombres ordonnés. La dichotomie consiste à partager l'ensemble en deux ensembles de même taille puis à tester auquel des deux ensembles le nombre appartient ; on recommence alors avec le nouvel ensemble et on reproduit ce processus **tant que** la taille de l'ensemble est supérieure à la précision souhaitée. Cette méthode permet à chaque étape de diviser la taille de l'ensemble contenant la solution, par exemple l'amplitude de l'intervalle, par deux. Donc en dix étapes, la taille est environ mille fois plus petite puisque $2^{10} = 1024$.

On souhaite afficher la table de multiplication par 13 :

```
i = 0
Début Tant que i <= 10
        Afficher 13*i
        i = i +1
Fin Tant que
```

Le programme en Python est le suivant :

```
i = 0
while i <= 10 :
        print('13 fois', i, '=', 13*i)
        i = i +1
```

L'itération se fait sur une suite d'entiers bien déterminés.

L'itération se fait sur une suite d'entiers bien déterminés.

Une autre manière de produire le même résultat est d'utiliser l'instruction "Pour" (For).

L'itération se fait sur une suite d'entiers bien déterminés.

Une autre manière de produire le même résultat est d'utiliser l'instruction "Pour" (For).

Cette instruction s'emploie pour répéter par exemple n fois une suite d'instructions lorsque n est connu à l'avance (pour déterminer un tableau de n valeurs d'une fonction, pour le calcul de n termes d'une suite, . . .).

L'algorithme est alors le suivant :

```
Entier i
Pour i variant de 0 à 10
        Afficher 13*i
Fin de Pour
```

Et l'implémentation en Python :

```
for i in range(0, 11) :
        print('13 fois', i, '=', 13*i)
```

Et l'implémentation en Python :

```
for i in range(0, 11) :
        print('13 fois', i, '=', 13*i)
```

On peut aussi écrire `for i in range(11)`, car s'il n'y a pas de premier argument pour `range`, le programme considère qu'il vaut 0.

Et l'implémentation en Python :

```
for i in range(0, 11) :
        print('13 fois', i, '=', 13*i)
```

On peut aussi écrire `for i in range(11)`, car s'il n'y a pas de premier argument pour `range`, le programme considère qu'il vaut 0.

De manière générale, on écrit :
`for i in range(début,fin,pas).`

Avec l'instruction `for i in range(100)`, la variable entière `i` prend successivement les valeur 0, 1, 2, . . . , 99, donc la boucle va s'effectuer 100 fois.

Avec l'instruction `for i in range(100)`, la variable entière `i` prend successivement les valeur 0, 1, 2, . . . , 99, donc la boucle va s'effectuer 100 fois.

Voici une boucle qui calcule et affiche les carrés des entiers pairs de 10 à 50 compris :

```
for i in range(10,51,2):
    print(i**2)
```

De manière générale, on écrit :

```
for variable in sequence :
        action1
        action2
        action3
```

La variable peut donc parcourir non seulement des entiers successifs mais aussi n'importe quel objet "itérable", (suite ordonnée d'éléments),

La variable peut donc parcourir non seulement des entiers successifs mais aussi n'importe quel objet "itérable", (suite ordonnée d'éléments), par exemple :

```
>>> for lettre in 'bonjour' :
        print(lettre, end =' ')
b o n j o u r
```

Une boucle peut être interrompue par l'instruction **break** :

```
>>> for x in range(0, 11) :
        if x == 5 :
            break
        print(x, end =' ')

0 1 2 3 4
```

Une boucle peut "sauter" une valeur avec l'instruction **continue** :

```
>>> for x in range(0, 11) :
        if x == 5 :
            continue
        print(x, end =' ')

0 1 2 3 4 6 7 8 9 10
```

De nombreux petits programmes souvent utilisés existent déjà et on les trouve dans des **bibliothèques**. Une bibliothèque pour Python (library en anglais, à ne pas confondre avec le mot français librairie qui se dit book-shop en anglais) est composée de modules ou packages.

De nombreux petits programmes souvent utilisés existent déjà et on les trouve dans des **bibliothèques**. Une bibliothèque pour Python (library en anglais, à ne pas confondre avec le mot français librairie qui se dit book-shop en anglais) est composée de modules ou packages.
Le module **math** contient des constantes et des fonctions mathématiques usuelles. Pour une documentation, consulter http ://docs.python.org/3/ et vérifier en particuler le nom et l'utilisation des fonctions ; par exemple la fonction mathématique ln est notée "log".

Pour utiliser le module **math**, il faut l'importer dans notre programme :

Pour utiliser le module **math**, il faut l'importer dans notre programme :

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
>>> math.log(3)
1.0986122886681098
>>> math.exp(1)
2.718281828459045
>>> math.sin(math.pi/2)
1.0
```

Afin de simplifier l'écriture des fonctions dans un programme, on peut choisir les éléments à importer du module **math** :

Afin de simplifier l'écriture des fonctions dans un programme, on peut choisir les éléments à importer du module **math** :

```
>>> from math import sqrt, log, sin
>>> sqrt(8)
2.8284271247461903
>>> log(5)
1.6094379124341003
>>> sin(1.57)
0.9999996829318346
```

On peut aussi utiliser l'instruction `import math as m` et le nouveau nom du module `math` dans notre programme est alors "m" ;

On peut aussi utiliser l'instruction `import math as m` et le nouveau nom du module `math` dans notre programme est alors "m" ; ou bien l'instruction `from math import*` qui importe tous les éléments du module ; ceci peut être pratique mais cela signifie aussi que notre programme est rempli d'un grand nombre de noms que nous n'utiliserons pas.

On peut aussi utiliser l'instruction `import math as m` et le nouveau nom du module `math` dans notre programme est alors "m" ; ou bien l'instruction `from math import*` qui importe tous les éléments du module ; ceci peut être pratique mais cela signifie aussi que notre programme est rempli d'un grand nombre de noms que nous n'utiliserons pas. Enfin, on peut aussi renommer les fonctions en écrivant par exemple :

```
>>> from math import log, e
>>> ln = log
>>> ln(e)
1.0
```

Note : les fonctions permettant de générer des nombres aléatoires sont dans le module **random** ;

Note : les fonctions permettant de générer des nombres aléatoires sont dans le module **random** ;
la fonction **random()** génère un flottant aléatoire dans l'intervalle $[0; 1[$, la fonction **randint(a,b)** avec a et b entiers génère un entier aléatoire n tel que a $\leq n \leq$ b.

Note : les fonctions permettant de générer des nombres aléatoires sont dans le module **random** ;
la fonction **random()** génère un flottant aléatoire dans l'intervalle $[0; 1[$, la fonction **randint(a,b)** avec a et b entiers génère un entier aléatoire n tel que a $\leq n \leq$ b.

Pour plus d'informations, documentation disponible sur http ://docs.python.org/3/library/random.html

Python permet d'effectuer des calculs avec des nombres complexes.

Python permet d'effectuer des calculs avec des nombres complexes. Le nombre imaginaire noté i en mathématiques s'écrit j en Python et le nombre complexe $2 - 3i$ s'exprime par (2-3j) en Python.

Python permet d'effectuer des calculs avec des nombres complexes. Le nombre imaginaire noté i en mathématiques s'écrit j en Python et le nombre complexe $2 - 3i$ s'exprime par (2-3j) en Python. Attention, le nombre complexe i s'exprime en Python par 1j (pas seulement j).

Exemples d'écriture :

```
>>> c1 = 5 - 3j
>>> c2 = 3 + 1j
>>> p = c1*c2
>>> print (p)
(18 - 4j)
>>> type (p)
<class 'complex'>
>>> abs(3 + 4j)
5.0
```

On peut également définir un complexe par sa partie réelle et sa partie imaginaire, ou déterminer les parties réelle et imaginaire d'un complexe donné ainsi que son conjugué :

On peut également définir un complexe par sa partie réelle et sa partie imaginaire, ou déterminer les parties réelle et imaginaire d'un complexe donné ainsi que son conjugué :

```
>>> a, b = 4, -5
>>> c = complex(a, b)
>>> c
(4-5j)
>>> c.real
4.0
>>> c.imag
-5.0
>>> c.conjugate()
(4+5j)
```

On peut également définir un complexe par sa partie réelle et sa partie imaginaire, ou déterminer les parties réelle et imaginaire d'un complexe donné ainsi que son conjugué :

```
>>> a, b = 4, -5
>>> c = complex(a, b)
>>> c
(4-5j)
>>> c.real
4.0
>>> c.imag
-5.0
>>> c.conjugate()
(4+5j)
```

Il existe aussi un module mathématique consacré aux nombres complexes, le module **cmath**.